Introduction to Unconventional Computing

1

Susan Stepney

Abstract
This chapter provides a broad overview of the field of unconventional computation, UComp. It includes discussion of novel hardware and embodied systems; software, particularly bio-inspired algorithms; and emergence and open-endedness.

1.1 Introduction

Before we start examining unconventional computing, it is useful to contrast it with *conventional* computing, also called classical computing, or Turing computing.

Convention is what is *generally done*, here: designing an algorithm that instructs the computer in precisely what it should do, one step at a time acting on digital data, to produce a well-defined output; coding that algorithm in a programming language like C or Python; running that program on typical commercial computer hardware such as a PC, tablet, smartphone, or even a supercomputer accessed through the cloud; and viewing the output as text or images.

*Un*conventional computing (UComp, also called non-standard computing) challenges one or more of these conventions. There are many aspects to challenge, and so there are many forms of UComp. The mathematician Stanislaw Ulam said:

> using a term like nonlinear science is … like referring to the bulk of zoology as the study of non-elephant animals.
>
> Campbell et al. (1985)

S. Stepney (✉)
Department of Computer Science, University of York, York YO10 5DD, UK
e-mail: susan.stepney@york.ac.uk

© Springer International Publishing AG 2017

E.R. Miranda (ed.), *Guide to Unconventional Computing for Music*,
DOI 10.1007/978-3-319-49881-2_1

The situation is analogous for UComp: one can argue that it is a much broader domain than conventional computation, although admittedly less deeply explored. In this chapter, we focus on three main areas of UComp:

- *hardware and embodiment*—computing is a physical process and can exploit the physical properties of material, from quantum systems to slime moulds, and more
- *software, particularly bio-inspiration*—biological systems can be modelled computationally and can be considered to be performing computation, but in ways different from our 'crisp', digital approaches
- *emergence and open-endedness*—the result of the computation is an emergent property of all the components in the system, not a single well-defined output, and the system has the possibility to generate novelty

For further reading on UComp, covering a wider range of aspects, see, for example, (Adamatzky 2017a, b; Cockshott et al. 2012; Copeland 2004; Stepney 2008, 2012a, b; Stepney et al. 2008). Given the unconventionality of some of the systems proposed and used, it is not necessarily clear whether those systems are indeed performing computation, or merely 'doing their own thing' non-computationally. Horsman et al. (2014a, b) address this issue.

1.2 Embodied in Unconventional Hardware

1.2.1 Quantum Computing

That computing is a physical process is demonstrated *par excellence* by the existence of quantum computing. Classical computing appears to be a highly abstract, mathematical process, that is independent of the laws of physics. However, quantum computing (Nielsen and Chuang 2000) has demonstrated that classical computing incorporates underlying assumptions about the physical properties of the computing system; quantum physics supports different computational models, with abilities that (almost certainly) exceed those of classical computers.

> Turing hoped that his abstracted-paper-tape model was so simple, so transparent and well defined, that it would not depend on any assumptions about physics that could conceivably be falsified, and therefore that it could become the basis of an abstract theory of computation that was independent of the underlying physics. 'He thought,' as Feynman once put it, 'that he understood paper.' But he was mistaken. Real, quantum-mechanical paper is wildly different from the abstract stuff that the Turing machine uses. The Turing machine is entirely classical, and does not allow for the possibility the paper might have different symbols written on it in different universes, and that those might interfere with one another. … That is why the resulting model of computation was incomplete.
>
> Deutsch (1997)

Classical desktop PCs are, of course, quantum devices; semiconductor transistors rely on quantum properties. However, these quantum physical properties are used to implement purely classical logic devices: Boolean switches. The computational *model* is classical.

The quantum circuit computational model is different, in that it relies on and exploits the specifically quantum properties of superposition and entanglement. Classical bits, that are either 0 or 1, are replaced by qubits (quantum bits) that can exist in a *superposition* of 0 and 1. Quantum algorithms using qubits can be run directly on a suitable quantum computer. Classical logic gates, such as AND, NAND, and NOR, which perform basic computations on bits, are replaced by quantum gates, such as 'controlled not' and 'Hadamard', which perform basic computation on qubits, and are furthermore reversible, in that they do not lose information and so can be run backwards. Quantum circuit models can also be *simulated*, often with considerable overhead (Feynman 1982), on a classical computer.

An alternative quantum computing approach is that of quantum annealing, as exploited by the commercial D-Wave quantum computer (Johnson et al. 2011; Lanting et al. 2014). The quantum annealing algorithm is discussed in Sect. 1.3.1.

1.2.2 Embedded Computing

Conventional computers are stand-alone devices, that are programmed, fed with data, and which output their symbolic results to the external user. The computer is a classical *disembodied* 'brain in a vat'.

Some computers are *embedded* in physical systems, usually to monitor or control them. Examples abound, from washing machines, cars, and robots to smart buildings and spaceships. Here, the input is directly from *sensors* (for light and sound, temperature, speed, heading, etc.), and the output is directly to *actuators* (sound production, heaters, motors, etc.). The underlying computation may nevertheless be relatively conventional: classical algorithms process input and produce output. The computer brain is separated from, although embedded in, the body.

Embedded systems do have one fundamental distinction from the purely classical model, however: they typically include feedback loops, where their actuators change the system or environment, and this change is fed back into the system through the sensors ready for the next round of computation. (Hence, such systems are necessarily real-time systems, as they must respond on a timescale dictated by environmental changes.) This 'guided' feedback structure is contrasted with the 'ballistic' model of non-interactive classical programs.

In both these cases, the *model* of classical computation is realised by some physical device, yet the immediate properties of that device are essentially unrelated to the computational model. The model is an abstract virtual layer of digital logic, with a large *semantic gap* to the underlying physical implementation: that is, there is a large difference between the form of description of the computational model and the form of description of its physical implementation.

This semantic gap between the computational model and its physical implementation is evidenced by the many physical substrates that can be engineered to support classical computation. All that is needed is for the material to be engineered to implement a digital switch, irrespective of the material it is made from. Hence, the classical model can be executed with silicon transistors; with radio valves, as in early presilicon computers; and even with brass wheels and cogs, as in the Babbage engine (Swade 1995).

1.2.3 Analogue Computing

Embodiment reduces the semantic gap between the computational model and the physical implementation. The physical properties of the computational device contribute directly to the computation being performed. Analogue computers are often an example of this embodiment. These devices function 'by analogy' to the way the problem functions.

For example, a particular electrical circuit can be built that behaves in a way analogous to a swinging pendulum: the oscillations of the electric voltage are analogous to the oscillations of the pendulum bob and so can be used to predict the pendulum's behaviour.

Another example is Monetary National Income Analogue Computer (MONIAC), built as a model of the UK's economy, where the flow of water is an analogue of the flow of money.

> The MONIAC was capable of making complex calculations that could not be performed by any other computer at the time. The linkages were based on Keynesian and classical economic principles, with various tanks representing households, business, government, exporting and importing sectors of the economy.
>
> Water pumped around the system could be measured as income, spending and GDP. The system was programmable, and experiments with fiscal policy, monetary policy and exchange rates could be carried out.
>
> (Reserve Bank Museum)

Analogue computers can be a special purpose (such as an orrery, computing planetary positions for one specific solar system), or they can be a general purpose (Rubel 1993; Shannon 1941). Programming can be a combination of designing the necessary circuit in terms of differential equations and then implementing that circuit in hardware, via a patch panel connecting basic electrical and electronic components in the general-purpose devices built in the 1950s and 1950s, or via a digital interface in the case of the more recent electronic field-programmable analogue arrays.

Analogue computation is embodied, because it depends on the actual physical properties of the circuit. Because most analogue computers use continuous variables, such as voltage or position, the word 'analogue' has also come to mean *continuous*, as opposed to the discrete, digital representations used in conventional computers.

1.2.4 Unconventional Substrates: *In Materio* Computing

Analogue computing uses a specific substrate where that substrate has known analogous behaviour to a specific problem in question. Unconventional substrates can also be used to perform computation intrinsic to their own behaviours: this is often called *in materio* computing. A multitude of complex substrates have been examined for computational properties: here, we survey a few of the more developed approaches.

One such approach is chemical substrates, designed to be reaction–diffusion systems. Chemicals diffuse through a medium, and chemicals react. The combination of these two processes can result in spatial patterns, including waves of activity (Turing 1952). Systems can be designed to solve specific problems, such as computing Voronoi diagrams (Adamatzky 1994) and navigating mazes (Steinbock et al. 1995). The waves and their interactions can be used to implement a wide variety of computations (Adamatzky et al. 2005).

Other complex materials, such as liquid crystals (Adamatzky et al. 2011; Harding and Miller 2004), carbon nanotubes (Dale et al. 2016; Mohid et al. 2015), and even conductive foam (Mills et al. 2006), have also been investigated as potential substrates for unconventional computing. The underlying rationale is that sufficiently complex materials can exhibit complex dynamics when provided with various inputs. Under certain circumstances, these dynamics can be interpreted and observed as computations performed on the inputs (Horsman et al. 2014b).

Biological materials are of special interest as unconventional computing substrates: they are complex[1] and highly evolved, and biological systems appear to perform intrinsic computation to some degree (Horsman et al. 2017).

Cells contain DNA, which contains genes. These genes code for proteins. Some proteins are *transcription factors*, binding to DNA and affecting how other genes are expressed, by inhibiting or promoting their expression. This complex interaction between genes and their proteins forms a regulatory network. Synthetic biology makes changes to DNA (modifying genes and their expression) in order to program small logic circuits into the gene regulatory system, usually in bacterial cells (Pease 2013). For example, a gene might be added that is ‘switched on’ (its expression is promoted) when two specific transcription factors are present. In this way, the engineered gene can be thought of as implementing a computational AND gate: it outputs a protein only when both input proteins are present. By linking several such gates together, a small logic circuit can be constructed. Hence, cell genomes can be engineered to compute functions of their inputs and to perform specific actions (expressing an output protein) depending on the result of that computation.

DNA is an interesting unconventional computing substrate, not just for its information storage and expression properties in living cells, but also for its construction abilities outside the cell. Short strands of DNA are relatively rigid and can

---

[1]The biological sketches given here are extremely simplified descriptions of highly complex processes.

be designed with ‘sticky ends’ that selectively glue to complementary ends on other strands. These can be used to implement self-assembling DNA ‘tiles’ (Winfree 2004) and compute a range of functions, by building microscopic patterned structures. Similar self-assembling computations can be programmed into macroscopic tiles made of other materials, where the tile assembly is mediated through mechanical or magnetic hooks.

At a larger biological scale, slime moulds can be used to perform certain computations (Adamatzky 2010). Their growth and movement behaviours are exploited for the specific computational purpose, including music production (Braund and Miranda 2015; Miranda and Braund 2017).

1.2.5 Embodied Environmental Interaction

Bringing together the concepts of embedded computing (computation controlling an active system, Sect. 1.2.2) and *in materio* computing (computation exploiting the material substrate properties, Sect. 1.2.4), we can get fully embodied computing: the material substrate of the system is being used computationally to help control the system itself.

An embodied computer is closely coupled with its environment in some way. The relevant environment might be any of: (i) the computational substrate, (ii) the system’s ‘body’, and (iii) the local external world. The aim often is to use the complexity of the coupling and environment to provide some of the computational power for the computing device.

When embodied in an unconventional computational substrate, the aim is for computation to be handed over to the specific physics of the device: the substrate’s behaviour naturally performs (some of) the desired computation, as explored in Sect. 1.2.4.

When embodied in a system ‘body’, such as a robot body, the aim is for some of the computation to be handled by the physical or mechanical properties of the body, rather than all aspects of the body’s behaviour being brute force computed as in classical embedded systems (Sect. 1.2.2). For example, in ‘passive dynamic walker’ robots, the entire process of locomotion is offloaded to the mechanical design (Collins et al. 2005).

When embodied in the local external world (Stepney 2007), the aim is for some of the computation to be handled by the properties of the world. An example from nature is stigmergy, where a mark left in the environment by an agent is later used to stimulate some other action, by that agent or another one. Ants laying pheromone trails that slowly evaporate are using stigmergy to communicate best paths to food to their nestmates. People writing shopping lists are using stigmergy to offload their memory burden. Other forms of embodiment offload a model building burden: robots build up a model of their environment to help navigation; some robot architectures ‘use the world as its own model’ (Brooks 1991).

1.2.6 Massively Parallel Substrates

One feature many of these unconventional substrates have is massive parallelism. Classical Turing computation is sequential: computational steps are taken one after the other. In parallel substrates, different portions of the material can be performing computations simultaneously, in parallel with other portions. Multicore PCs have a few tens of processors acting together; massively parallel devices have thousands, millions, or more parts acting in parallel. For example, slime moulds and reaction–diffusion systems are massively parallel.

Cellular automata, familiar through the example of Conway’s Game of Life (Gardner 1970), are a well-known massively parallel computational model. A large grid of very simple computing devices operate in parallel; each simple grid element communicates with its nearest neighbours to decide how to behave. All grid elements have the same program, but behave differently due to the differing states of their neighbourhoods. The time behaviour of 2D CAs, like the Game of Life, is typically presented as animations. The time behaviour of 1D CAs can be visualised statically (see Fig. 1.1). Suitably designed CAs can perform any classical computation (Rendell 2002) and can generate complex and beautiful patterns (Adamatzky and Martinez 2016; Owens and Stepney 2010). CAs are usually implemented using classical computers, where their massive parallelism is only simulated.

CAs live in a discrete space. Field computing assumes a continuous space and computes with combinations of mathematical fields to produce dynamic patterns (Beal and Viroli 2015).

1.2.7 Programming Unconventional Materials

Classical computing has a programming model: a process for designing instructions for the computer so that it will perform the desired computation. These instructions are given in a *high-level language* that provides useful abstractions far removed from the low-level bits and logic gates provided by the underlying physical implementation. For the most part, unconventional substrates have no such model, or even if there is a model [such as differential equations, CAs, field computing, or reservoir computing (see later)], there are few or no equivalents of high-level languages.

An alternative to systematic construction (programming) is *search*. Rather than constructing a particular program, one hunts through possible programs until one finds an acceptable solution. Bio-inspired search is often used, as discussed in the following section.

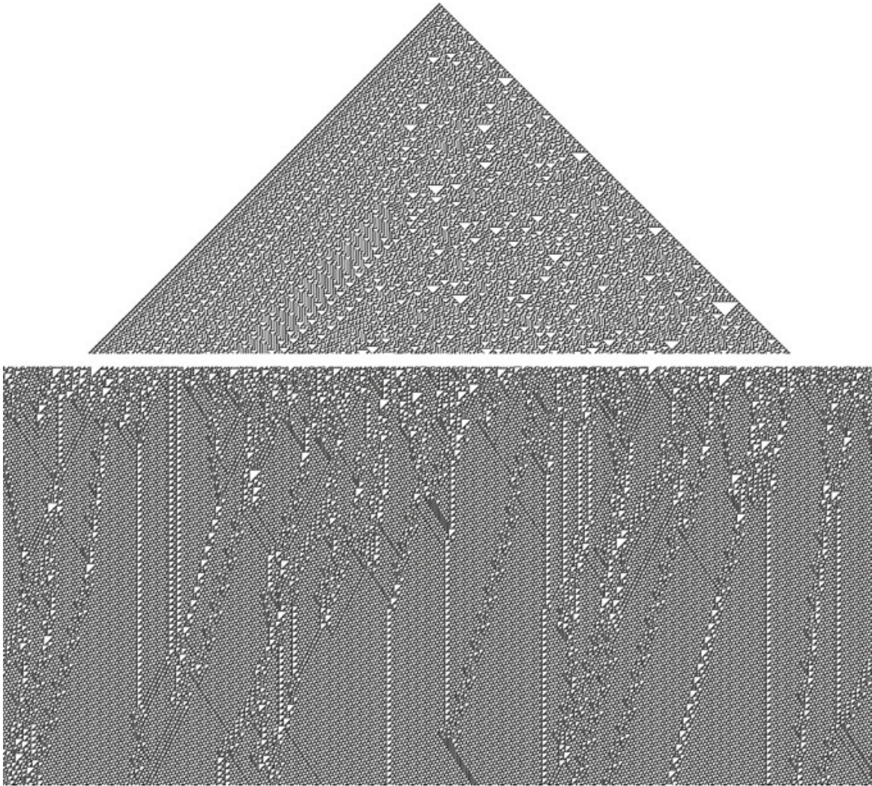

Fig. 1.1 Visualisation of the time evolution of a one-dimensional CA, with the grid cells arranged in a line. The CA's behaviour is shown over multiple timesteps. Each *horizontal line* shows the state of the line of grid cells; *subsequent lines* down the page correspond to subsequent timesteps. *Top* the CA 'rule 30' runs from an initial state that has one *black cell* and all the others *white*. *Bottom* the CA 'rule 110' runs from an initial state where each cell is randomly either *black* or *white*

1.3 Computation Inspired by Nature

This section deals with unconventional algorithms. Although often implemented on a classical computer in a standard programming language, these algorithms are inspired by the way natural processes work, which are often fuzzy, inexact, and suboptimal, in contrast to 'crisp', exact classical algorithms.

1.3.1 Inspired by Physics

The simulated annealing search algorithm (Kirkpatrick et al. 1983) is inspired by the physical processes that occur when a metal is slowly cooled so that it reaches it

ground state of minimum energy. In the algorithm, the minimum energy state corresponds to the desired solution, and a temperature analogue is used to control movement around the search space.

Quantum circuit algorithms (Sect. 1.2.1) are the most obvious form of computation inspired by nature, the physical laws of quantum mechanics. Quantum circuit algorithms include Shor's factorisation algorithm (Shor 1997) and Grover's search algorithm (Grover 1996).

Quantum annealing (Kadowaki and Nishimori 1998; Santoro and Tosatti 2006) works in a different way from quantum circuit algorithms. A quantum state is gradually ('adiabatically') changed, and quantum tunnelling allows the system to find the ground state. It is a quantum analogue of simulated annealing, exploiting quantum tunnelling through barriers in the landscape, rather than thermal energy to jump over barriers. In simulated annealing, temperature and energetic analogues are *simulated* in a digital computer; in quantum annealing, *physical* energetics and quantum tunnelling are exploited in a quantum computer.

These examples are inspired by branches of physics. Most nature-inspired algorithms are, however, inspired by a wide range of *biological* processes.

1.3.2 Population-Based Computation

Population-based computation draws its inspiration from a wide range of biological systems. In population-based computation, there is usually a population of 'organisms' or 'cells' working in competition or collaboration to *search* for a sufficiently good solution. The populations and their search process include evolutionary algorithms, based on a population of creatures competing for survival (Mitchell 1996), immune algorithms, based on a population of antibodies competing to recognise an intruder (de Castro and Timmis 2002), particle swarm optimisation, based on flocks of birds cooperating in the search for food (Kennedy and Eberhart 1995), and ant colony optimisation, based on a nest of ants cooperating to find a short path (Dorigo et al. 1996). Cooperative *swarm intelligence* algorithms, based on flocks and social insect swarms, are also used in other cooperative behaviour applications, such as a swarm of small robots cooperating to perform a particular task.

The population-based search algorithms differ in how much inspiration they take from nature. Underlying them all is a similar form of process (Newborough and Stepney 2005): each member of the population is at a position in a 'fitness landscape'; fitter members of the population have more progeny; and progeny resemble their parents (hence fitter solutions survive) with some variation (hence the fitness landscape is explored). Differences between members of this class of algorithms are mainly in the variation stage, differing in exactly how the next generation is created from the current one and hence in exactly how the fitness landscape is explored.

The key to using these algorithms successfully is to balance exploration (moving around the landscape) with exploitation (sticking with, and improving on, discovered solutions). Too much exploration and the search do not converge (the search is

essentially random); too much exploitation and it converge on a local, but not global, optimum (the search gets stuck on a local peak and never finds Everest).

The inspiring biological systems evolve in parallel: each member of the population lives, reproduces, and dies alongside the others. The resulting algorithms are usually sequentialised.

Evolutionary search can be exploited to program unconventional substrates. The main form of this is *evolution in materio* (Broersma et al. 2017; Miller and Downing 2002; Miller et al. 2014). Configuration voltages are evolved such that, when applied to the specific material, it performs the desired computation.

Key to evolutionary search and its brethren is the fitness function (alternatively called the cost function, or the affinity, depending on the specific algorithm type).

In well-defined optimisation problems, the fitness function is relatively easy to define. It is known what is a 'good' solution and how to quantify it so that solutions can be ranked one better than another. However, for problems where the evaluation of solutions requires human judgement, making an algorithmic ranking difficult or impossible to define, such as judging visual art or music, it may be necessary to include a 'human in the loop' to act as the (subjective) fitness function. In such cases, small populations (tens rather than hundreds) and few generations (hundreds rather than thousands) tend to be used, because of user fatigue. This can be done to provide a single good result, or used in an interactive manner, such as to generate music where the user's subjective fitness evaluation may change as the composition develops (Hickinbotham and Stepney 2016).

Alternatively, in these more creative situations, a fitness-function-free *novelty search* (Lehman and Stanley 2011) can be used. This prioritises exploration over exploitation, by requiring new solutions simply to be *different* from current ones. This supports an open-ended (Banzhaf et al. 2016) exploration of the possibility space, rather than trying to home in on a specific ill-defined optimal solution.

1.3.3 Network-Based Computation

Network-based computation draws on a rich suite of biological processes. A network comprises a collection of nodes joined by edges. A biological network might be physical (a neural network, where the nodes are the soma or cell body, and the edges are the axons and dendrites that connect neurons together). However, the network is more often abstract: the nodes are typically physical objects, but the edges are abstractions of different kinds of interaction. Examples include genetic regulation networks (the regulatory interaction between genes via their expressed proteins), metabolic networks (the interactions between metabolic molecules, mediated by enzymes), signalling networks (interaction pathways as signal molecules propagate from the outside to the interior of a cell, mediated by proteins), food webs (who eats whom), and social networks (who are friends with whom). Most work has focussed on neural-inspired models (Callan 1999). Although these networks are diverse, some with physical, some with virtual connections, they have common underlying properties (Lones et al. 2013).

Many network-based algorithms are *learning* algorithms. The network is presented with 'training' data. The algorithm adjusts network parameters (often weights in the nodes and edges) in response to the data so that, when it see the same or similar data in the future, it outputs some relevant response. So it is trained to 'recognise' patterns in data.

As an alternative to using a learning algorithms to train a network, it is possible to use evolutionary search to find good weights. One well-established approach to evolving networks is NeuroEvolution of Augmenting Topologies, or NEAT (Stanley and Miikkulainen 2002), which evolves both the weights and the network topology. Compositional pattern-producing networks (CPPNs) can be used to produce spatial patterns (Stanley 2007). These two techniques, NEAT and CPPNs, have been combined in the HyperNEAT approach (Stanley et al. 2009), allowing the evolution of pattern-producing networks.

A recent development with using neural networks to produce haunting images is Google's Deep Dream (Mordvintsev et al. 2015). There are two steps. First, a neural network is trained to recognise and classify images. It is then given a new image and asked to recognise features in it from its training (that cloud looks a little like a face); these features are enhanced slightly (to make the cloud look a little more like a face); and the enhanced image fed back into the network (the face is even more recognisable and so gets further enhanced). This *iterative feedback* process can lead to weird and wonderful images.

The choice of training images affects the resultant pictures. The original work seems to have used training images of dogs and eyes, leading to eerie surreal pictures with dogs and eyes everywhere. It may be the first example of artificial pareidolia (Roth 2015), that until now purely human tendency to see faces where there are no actual faces.

The Deep Dream approach trains on many images. Using a related approach that trains on only a single images allows the production of one picture in the style of another (Gatys et al. 2015), resulting in less surreal, more 'artistic' images; see Fig. 1.2.

1.3.4 Generative Computation

The processes of iteration and feedback, components of the Deep Dream image production (Sect. 1.3.3), are also key components of generative computing. The result is generated, or constructed, or developed, or 'grown', through a repeated series of steps. Each step uses the same growth rules, but in a context changed by the previous growth step. This changed context, fed back into the computation, potentially produces new details at each step.

Artificial Chemistries (Banzhaf and Yamamoto 2015; Dittrich et al. 2001) are algorithms inspired by the way natural chemistry assembles atoms into large complex molecules through reactions. Computational analogues of atoms, molecules, and reaction rules are defined; these analogues can have similar properties to

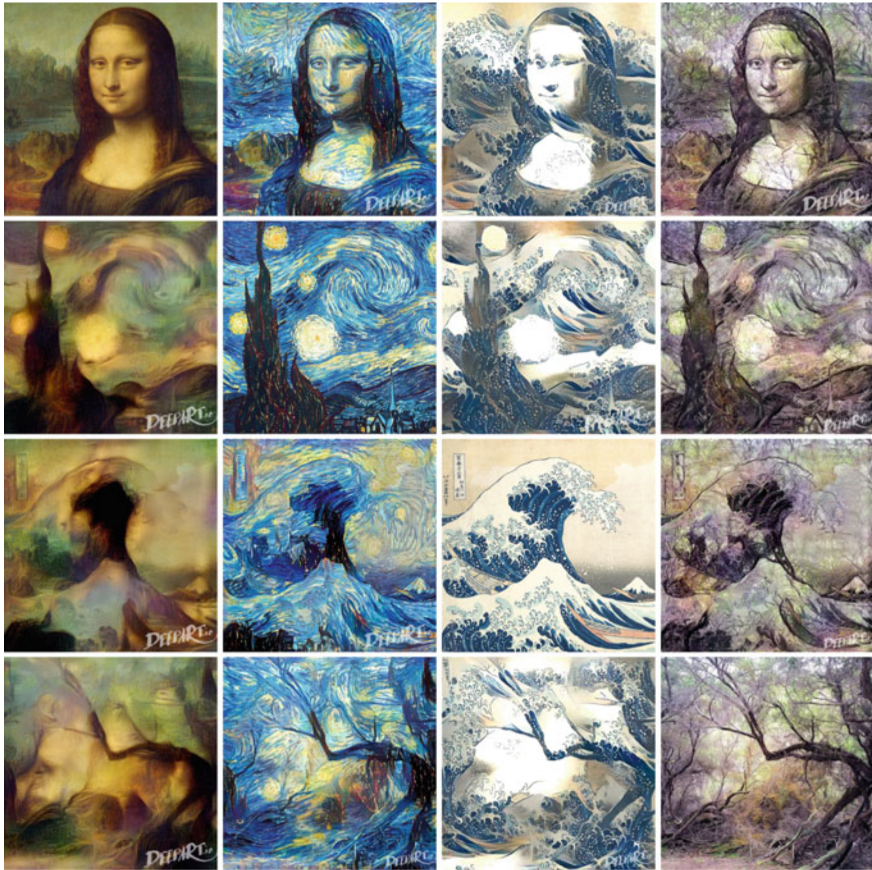

Fig. 1.2 One picture in the style of another (images generated at deepart.io). The four original pictures (shown down the main diagonal) are a portion of: da Vinci's Mona Lisa, van Gogh's The Starry Night, Hokusai's The Great Wave, woodland scene photograph by the author. Each row shows one original in the style of all the other pictures; each column shows all the pictures in the style of one original

natural atoms, or might be related by only a tenuous analogy. These components are combined using the reaction rules to generated complex structures.

Lindenmeyer's L-Systems (Prusinkiewicz and Lindenmayer 1990) were originally invented to model plant growth. They are one of a class of rewriting rules, or generative grammars, where parts of a structure are successively rewritten ('grown') according to a set of grammar rules. If the structures being rewritten are graphical components, the resulting L-System can mimic plant growth, or geometrical constructions. The rewriting rules can be applied probabilistically, to provide a natural irregularity to the constructions (see Fig. 1.3). The rewritten structures can be types other than graphical components, including musical notes and phrases (Worth and Stepney 2005).

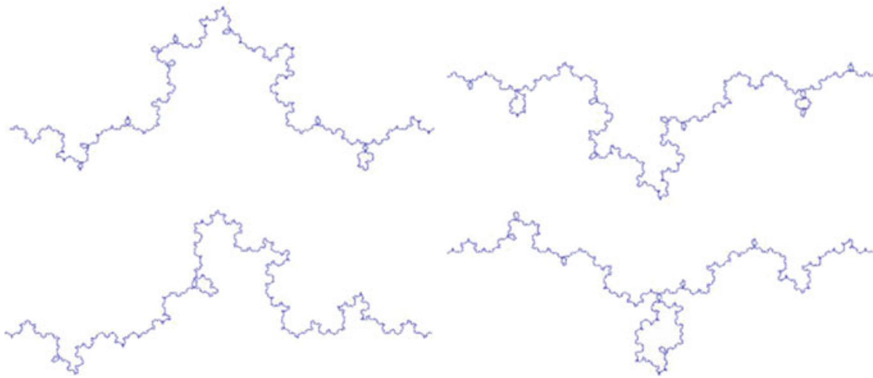

Fig. 1.3 A stochastic L-system: the same generative rules, applied probabilistically, produce a family of related pictures

The ideas of evolutionary algorithms can be combined with growth rules to form an 'evo-devo' (evolution of development), or morphogenetic, system. In the simplest cases, a 'seed' (initial state) is evolved and then 'grown' into an adult form. In more advanced cases, the growth rules may also be evolved. This approach allows a sophisticated set of related adult forms, varied by changing parameters, starting conditions, and random seeds. Such complex forms would be harder to evolve directly, as the growth process can coordinate development of related structures, for example, making a regular pattern of legs (Hornby 2004). Evo-devo approaches can produce remarkably 'organic'-looking images (Doursat et al. 2012; Todd and Latham 1992).

It has been suggested that a combination of evo-devo systems and synthetic biology (evolving rules to program into biological cells), and other UComp approaches, could be exploited to grow architectural structures (Armstrong 2015, 2017; Stepney et al. 2012).

1.3.5 Dynamical Systems Computation

The classical Turing model has a single output when the computation halts. That output may have a complex structure, but is essentially a static view of the final state of the computation. Considering a computation as being a dynamical system, as being about the computed movement of the system through its state space, allows a more dynamical view of the process and its outputs over time (Stepney 2012a). It also encompasses interactive computing (Wegner 1997), where inputs are given to the system during its execution, inputs that may be a feedback response to previous outputs.

Cellular automata (Sect. 1.2.6) are an example of a discrete dynamical system. A related system is that of random Boolean networks (RBNs). While CAs have a

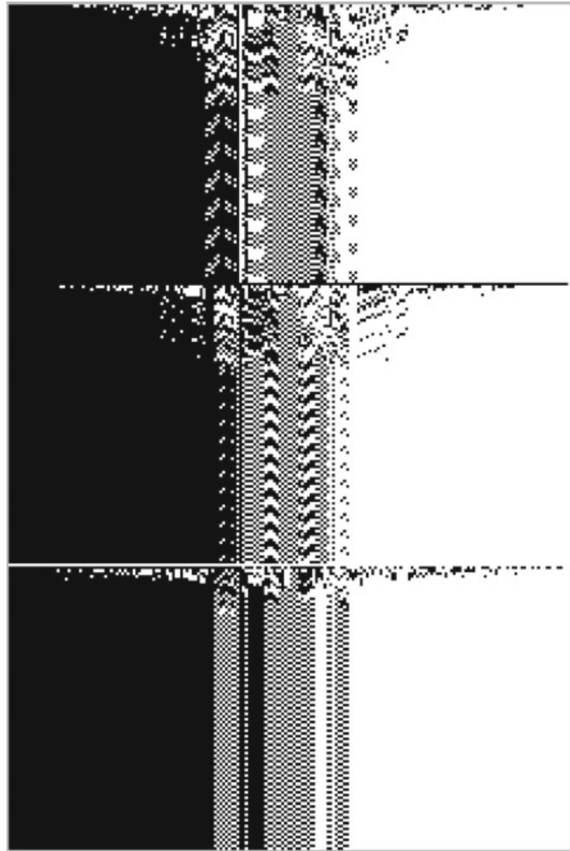

Fig. 1.4 Visualisation of the time evolution of a typical RBN where each cell receives input from two of its neighbours. This RBN has 200 randomly connected cells, which are arranged in an order that exposes the dynamical structure. Three different initial conditions of this network illustrate three different final behaviours, that is three different attractors

uniform grid of cells, and each cell follows the same rule, RBNs have their nodes wired together randomly, and each cell has its own randomly chosen rule. Despite all this randomness, certain RBNs display well-defined patterns of dynamics (see Fig. 1.4).

Dynamical systems can be described as traversing trajectories through their state spaces (space of all possible configurations). Trajectories may wander throughout the entire state space, or they may converge to certain smaller regions of the state space, called *attractors*. [For more detail, see Stepney (2012a).] These attractors form a higher level emergent structure to the system. Some attractors are merely single points in the state space. A pendulum, no matter where started, will end up in the same final resting state: it has a single point attractor. Some attractors are cycles: the system repeatedly cycles through the same sequence of states. The RBN in Fig. 1.4 demonstrates three different cyclic attractors. Some attractors are *strange*: although many trajectories are attracted to a given region of the state space, within

that region the trajectories diverge from each other. Such systems display *sensitive dependence on initial conditions*: two systems started in very similar states will, after a while, be displaying very different behaviours (albeit in the same restricted part of their state space). The detailed structure of a strange attractor is usually *fractal* (Mandelbrot 1997).

Reservoir computing is a model of a form of dynamical computation. It combines three methods for creating and training neural networks: echo state networks (Jaeger 2001), liquid state machines (Maass et al. 2002), and the backpropagation-decorrelation online learning rule (Steil 2004). The underlying model is a neural network (Sect. 1.3.3), although set up and used differently from the more traditional classifier style of these networks. Node connections and weights are set up randomly, and only the output weights are trained. Reservoirs can be simulated with classical computers or implemented in novel hardware. It can be used as a model for a variety of forms of novel hardware (Dale et al. 2017). The hardware itself can be configured to form a 'better' reservoir, which acts as a low-level virtual machine for programming the hardware (Dale et al. 2016). Neural network-based dynamical systems can also be used to underpin other UComp paradigms (Stovold and O'Keefe 2017).

Music is, of course, fundamentally temporal and multidimensional (pitch, timbre, etc.). A multidimensional dynamical system may form a suitable computational basis for generating novel music and musical styles.

1.4 Beyond Classical Thinking

1.4.1 Emergent Computation

Emergence is variously defined, but has the idea of something coming out of a system that was not explicitly or deliberately put into, or part of, the system.

Aristotle (1924) has one of the earliest definitions of what we now call emergence: 'things which have several parts and in which the totality is not, as it were, a mere heap, but the whole is something beside the parts', now commonly phrased as *the whole is more than the sum of its parts*. Anderson (1972) coined the phrase 'More is different' to describe emergence in physical systems.

One feature is that the concepts used to describe an emergent system are somehow different from, even at a higher level than, the concepts used to describe the underlying system. For example, consider cellular automata (Sect. 1.2.6) and Conway's Game of Life in particular. The underlying system is simply a grid of cells that are either 'alive' or 'dead' and which evolve through time according to a simple rule. From this underlying system emerge blocks, gliders, and a whole host of other macroscopic patterns.

Classical computation with its classical algorithms is not considered emergent in this sense.[2] The functionality that comes out is precisely what was programmed in (unless there are bugs). Something like music, however, can be considered as emergent from the series of individual notes and other sounds that make it up.

Many forms of UComp support forms of emergent computation, where the resulting global pattern or dynamics is an emergent or implicit property of the system, rather than having been explicitly programmed. In particular, many dynamical systems (Sect. 1.3.5) have such kinds of emergent properties, including the structure of their attractors.

1.4.2 Novelty and Open-Ended Computation

Emergence and other forms of novelty are desirable properties of art. Art itself should be 'open-ended', in that new forms are constantly possible.

Open-endedness, like emergence, is variously defined. Most definitions come down to 'the continual production of novelty'. Banzhaf et al. (2016) define types of novelty and open-endedness with respect to some domain of interest. *Novelty* in an observed system is classified into three kinds, with increasing complexity:

1. Variation: novelty within an instance of the domain. Variation changes an instance of the domain, such as a new painting in a particular style, without changing the domain itself. Variation explores a predefined state space, producing new values of existing ideas (in Fig. 1.2, each new entry in a particular column is a variation in a particular style).
2. Innovation: novelty that changes the domain. Innovation changes the domain: for example, by adding a new style of painting (e.g. expressionism was an innovation in this sense). Innovation changes the combinatorics and the size/structure of the domain, thereby growing the possibilities of variation (in Fig. 1.2, each new column is a novel style).
3. Emergence: novelty that changes the type of the domain. Emergence changes the type of the domain: a change that adds a new concept or dimension to the domain. For example, adding a third dimension, or movement, or sound, to pictures is the emergence of a new type of art.

Open-endedness is then defined as the ability to continually produce innovative or emergent events. Variation alone is not enough: there needs to be some kind of 'breaking out of the box', and then breaking out of the new box, and so on.

[2]Although certain properties of classical systems, such as security and performance, can be considered to be emergent, this emergence is one of the things that makes such properties hard to engineer.

1.5 Conclusions

Unconventional hardware and unconventional software allow us to re-examine concepts, constraints, and assumptions of computation. Unconventional hardware that supports dynamical, growing, evolving, and feedback processes offers a vast array of possible devices and ways of thinking about computing. UComp provides a rich source of novelty, which can be used for many applications, including the creation of novel artworks.

1.6 Questions

1. What analogue and digital properties does music have?
2. What are the similarities and differences between a musical score and a computer program?
3. Music has temporal structure, but can be represented spatially, such as in a score. How many different ways can you find to map spatial patterns of UComp substrates into spatial musical representations? What properties does this suggest the UComp spatial patterns should exhibit?
4. Discuss ways you could map a population-based search (Sect. 1.3.2) onto a musical structure.
5. This chapter suggests that searching for aesthetic pieces, such as music, requires a 'human in the loop' to act as the fitness function (Sect. 1.3.2). What are the advantages and disadvantages of this? How might at least some of the task be automated?
6. Discuss ways you could map the spatial and temporal structure of a network-based system (Sect. 1.3.3) onto a musical structure.
7. Generative computation (Sect. 1.3.4) is used to grow mainly spatial structures. What are the similarities and differences in growing a temporal structure?
8. Music has been claimed to have a fractal structure. What is meant by this? What more is needed for music, in addition to such structure?
9. Describe a range of existing forms of variation, innovation, and emergent novelty as they occur in music (see Sect. 1.4.2). How might such novelties be built into a UComp music system?
10. Can music be exploited as a form of UComp?

References

Adamatzky, A. (1994). Constructing a discrete generalized Voronoi diagram in reaction-diffusion media. *Neural Networks World, 40*(6), 635–644.

Adamatzky, A. (2010). *Physarum machines: Computers from slime mould*. World Scientific.

Adamatzky, A. (Ed.). (2017a). *Advances in unconventional computing, volume 1: Theory*. Berlin: Springer.

Adamatzky, A. (Ed.). (2017b). *Advances in unconventional computing, volume 2: Prototypes, models and algorithms*. Berlin: Springer.

Adamatzky, A., & Martinez, G. J. (Ed.). *Designing beauty: The art of cellular automata*. Berlin: Springer.

Adamatzky, A., De Lacy Costello, B., & Asai, T. (2005). *Reaction-diffusion computers*. London: Elsevier.

Adamatzky, A., Kitson, S., De Lacy Costello, B., Matranga, M. A., & Younger. D. (2011). Computing with liquid crystal fingers: Models of geometric and logical computation. *Physical Review E: Statistical, Nonlinear, Biological, and Soft Matter Physics*, *840*(6), 0 061702.

Anderson, P. W. (1972). More is different. *Science*, *1770*(4047), 393–396.

Aristotle. (1924). *Metaphysics, book VIII,* 350 BCE (trans. by W. D. Ross*, Aristotle's metaphysics*), 2 vols. Oxford: Oxford University Press.

Armstrong, R. (2015). How do the origins of life sciences influence 21st century design thinking? In *ECAL 2015* (pp. 2–11). Cambridge: MIT Press.

Armstrong, R. (2017). Experimental architecture and unconventional computing (pp. 773–804). In Adamatzky2017v2.

Banzhaf, W., & Yamamoto, L. (2015). *Artificial chemistries*. Cambridge. MIT Press.

Banzhaf, W., Baumgaertner, B., Beslon, G., Doursat, R., Foster, J. A., McMullin, B., … & White, R. (2016). Defining and simulating open-ended novelty: Requirements, guidelines, and challenges. *Theory in Biosciences*, *135*(3), 131–161.

Beal, J., Viroli, M. (2015). Space-time programming. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, *3730*(2046).

Braund, E., & Miranda, E. (2015). Music with unconventional computing: Towards a step sequencer from plasmodium of *Physarum polycephalum*. In *EvoMusArt 2015*, volume 9027 of LNCS (pp. 15–26). Berlin: Springer.

Broersma, H., Miller, J. F., & Nichele, S. (2017). Computational matter: Evolving computational functions in nanoscale materials (pp. 397–428). In Adamatzky2017v2.

Brooks, R. A. (1991). How to build complete creatures rather than isolated cognitive simulators (pp. 225–239). In *Architectures for intelligence: 22nd Carnegie Mellon Symposium on Cognition*.

Callan, R. (1999). *The essence of neural networks*. New York: Prentice Hall.

Campbell, D., Farmer, D., Crutchfield, J., & Jen, E. (1985). Experimental mathematics: The role of computation in nonlinear science. *Communications of ACM*, *280*(4), 374–384.

Cockshott, P., Mackenzie, L. M., & Michaelson, G. (2012). *Computation and its limits*. New York: Oxford University Press.

Collins, S., Ruina, A., Tedrake, R., & Wisse, M. (2005). Efficient bipedal robots based on passive-dynamic walkers. *Science*, *3070*(5712), 1082–1085.

Copeland, B. J. (2004). Hypercomputation: Philosophical issues. *Theoretical Computer Science*, *3170*(1–3), 251–267.

Dale, M., Miller, J. F., Stepney, S., & Trefzer, M. A. (2016). Evolving carbon nanotube reservoir computers. In *UCNC 2016*, volume 9726 of LNCS (pp. 49–61). Berlin: Springer.

Dale, M., Miller, J. F., & Stepney, S. (2017). Reservoir computing as a model for *in materio* computing (pp. 533–571). In Adamatzky2017v1.

de Castro, L. N., & Timmis, J. (2002). *Artificial immune systems: A new computational intelligence approach*. Berlin: Springer.

Deutsch, D. (1997). *The fabric of reality*. Penguin.

Dittrich, P., Ziegler, J., & Banzhaf, W. (2001). Artificial chemistries—A review. *Artificial Life*, *70* (3), 225–275.

Dorigo, M., Maniezzo, V., & Colorni, A. (1996). Ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, *260*(1), 29–41.

Doursat, R., Sayama, H., & Michel, O. (Eds.), *Morphogenetic engineering: Towards programmable complex systems*. Berlin: Springer.

Feynman, R. P. (1982). Simulating physics with computers. *International Journal of Theoretical Physics*, *210*(6–7), 467–488.

Gardner, M. (1970). The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, 120–123, October 1970.

Gatys, L. A., Ecker, A. S., & Bethge, M. (2015). A neural algorithm of artistic style. *CoRR*, abs/1508.06576, arxiv.org/abs/1508.06576.

Grover, L. K. (1996). A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (pp. 212–219), ACM.

Harding, S., & Miller, J. F. (2004). Evolution in materio: A tone discriminator in liquid crystal. In *Congress on Evolutionary Computation (CEC2004)* (Vol. 2, pp. 1800–1807).

Hickinbotham, S., & Stepney, S. (2016). Augmenting live coding with evolved patterns. In *EvoMusArt* 2016 (vol. 9596, pp. 31–46). Berlin: Springer, LNCS.

Hornby, G. S. (2004). Functional scalability through generative representations: The evolution of table designs. *Environment and Planning. B: Planning and Design, 310*(4), 569–587.

Horsman, C., Stepney, S., & Kendon, V. (2014a). *When does an unconventional substrate compute?* UCNC 2014 Poster Proceedings, University of Western Ontario Technical Report 758.

Horsman, C., Stepney, S., Wagner, R. C., & Kendon. V. (2014b). When does a physical system compute? *Proceedings of the Royal Society A*, *4700*(2169), 182.

Horsman, D., Kendon, V., Stepney, S., & Young, P. (2017). Abstraction and representation in living organisms: When does a biological system compute? In G. Dodig-Crnkovic, & R. Giovagnoli (Eds.), *Representation and reality: Humans, animals, and machines*. Berlin: Springer (in press).

Jaeger, H. (2001). The "echo state" approach to analysing and training recurrent neural networks. GMD Technical Report 148, German National Research Center for Information Technology, Bonn, Germany, 2001 (with an Erratum note, 2010).

Johnson, M. W., et al. (2011). Quantum annealing with manufactured spins. *Nature*, *4730*(7346), 194–198.

Kadowaki, T., & Nishimori, H. (1998). Quantum annealing in the transverse Ising model. *Physical Review E*, *580*(5), 5355–5363.

Kennedy, J., & Eberhart, R. (1995). Particle swarm optimization. In *IEEE International Conference on Neural Networks 1995* (vol. 4, pp. 1942–1948).

Kirkpatrick, S., Gelatt, C. D. Jr, & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *2200*(4598), 671–680.

Lanting, T., et al. (2014). Entanglement in a quantum annealing processor. *Physical Review X*, *40* (2), 021041.

Lehman, J., & Stanley, K. O. (2011). Abandoning objectives: Evolution through the search for novelty alone. *Evolutionary Computation 1*, *190*(2), 189–223.

Lones, M. A., Turner, A. P., Fuente, L. A., Stepney, S., Caves, L. S. D., & Tyrrell, M. (2013). Biochemical connectionism. *Natural Computing*, *120*(4), 453–472.

Maass, W., Natschläger, T., & Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural Computation*, *140*(11), 2531–2560.

Mandelbrot, B. B. (1997). *The fractal geometry of nature*. Freeman.

Miller, J. F., & Downing, K. (2002). Evolution in materio: Looking beyond the silicon box. In *Proceedings of NASA/DoD Conference on Evolvable Hardware, 2002* (pp. 167–176).

Miller, J. F., Harding, S. L., & Tufte, G. (2014). Evolution-in-materio: Evolving computation in materials. *Evolutionary Intelligence*, *70*(1), 49–67.

Mills, J. W., Parker, M., Himebaugh, B., Shue, C., Kopecky, B., & Weilemann, C. (2006). “Empty space” computes: The evolution of an unconventional supercomputer. In *Proceedings of the 3rd Conference on Computing Frontiers*, CF ’06 (pp. 115–126).

Miranda, E. R., & Braund, E. (2017). Experiments in musical biocomputing: Towards new kinds of processors for audio and music (pp. 739–761). In Adamatzky2017v2.

Mitchell, M. (1996). *An introduction to genetic algorithms*. Cambridge: MIT Press.

Mohid, M., Miller, J. F., Harding, S. L., Tufte, G., Massey, M. K.,et al. (2015). Evolution-in-materio: Solving computational problems using carbon nanotube–polymer composites. *Soft Computing* 1–16.

Mordvintsev, A., Olah, C., & Tyka, M. (2016). *Inceptionism: Going deeper into neural networks*, June 2015. http://ifundefinedselectfontresearch.googleblog.com/2015/06/inceptionism-going-deeper-into-neural.html. Accessed 14 June 2016.

Newborough, J., & Stepney, S. (2005). A generic framework for population-based algorithms, implemented on multiple FPGAs. In *ICARIS 2005*, volume 3627 of *LNCS* (pp. 43–55). Berlin: Springer.

Nielsen, M. A., & Chuang, I. L. (2000). *Quantum computation and quantum information*. Cambridge: Cambridge University Press.

Owens, N., & Stepney, S. (2010). The game of life rules on Penrose tilings. In A. Adamatzky (Ed.), *Game of life cellular automata* (pp. 331–378). Springer, Berlin.

Pease, R. (2013). How to turn living cells into computers. *Nature News*, February 2013.

Prusinkiewicz, P., & Lindenmayer, A. (1990). *The algorithmic beauty of plants*. Berlin: Springer.

Rendell, P. (2002). Turing universality of the Game of Life. In Andrew Adamatzky, editor, *Collision-Based Computing*, pages 513–539. Berlin: Springer.

Reserve Bank Museum. (2016). The MONIAC, a pioneering econometric computer. ifundefinedselectfont www.rbnzmuseum.govt.nz/activities/moniac. Accessed May 2, 2016.

Roth, B. (2015). Deepdream algorithmic pareidolia and the hallucinatory code of perception, October 2015. ifundefinedselectfont http://doorofperception.com/2015/10/google-deep-dream-inceptionism/. Accessed June 14, 2016.

Rubel, L. A. (1993). The extended analog computer. *Advances in Applied Mathematics*, *140*(1), 39–50.

Santoro, G. E., & Tosatti, E. (2006). Optimization using quantum mechanics: Quantum annealing through adiabatic evolution. *Journal of Physics A*, *390*(36), R393.

Shannon, C. E. (1941). Mathematical theory of the differential analyzer. *Journal of Mathematics and Physics*, *200*(1–4), 337–354.

Shor, P. W. (1997). Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, *260*(5), 1484–1509.

Stanley, K. O. (2007). Compositional pattern producing networks: A novel abstraction of development. *Genetic Programming and Evolvable Machines*, *80*(2), 131–162.

Stanley, K. O., & Miikkulainen, R. (2002). Evolving neural networks through augmenting topologies. *Evolutionary Computation*, *100*(2), 99–127.

Stanley, K. O., D’Ambrosio, D. B., & Gauci, J. (2009). A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, *150*(2), 185–212.

Steil, J. J. (2004). Backpropagation-decorrelation: Online recurrent learning with o(n) complexity. In *2004 IEEE International Joint Conference on Neural Networks* (vol. 2, pp. 843–848). IEEE.

Steinbock, O., Tóth, A., & Showalter, K. (1995). Navigating complex labyrinths: Optimal paths from chemical waves. *Science*, *2670*(5199), 868–871.

Stepney, S. (2007). Embodiment. In D. Flower, & J. Timmis (Eds.), *In silico immunology*, pp.265–288. Berlin: Springer.

Stepney, S. (2008). The neglected pillar of material computation. *Physica D*, *2370*(9), 1157–1164.

Stepney, S. (2012a). Non-classical computation: A dynamical systems perspective. In G. Rozenberg, T. Bäck, & Kok, J. N. (Eds.), *Handbook of natural computing* (pp. 1979–2025). Berlin: Springer.

Stepney, S. (2012b). Programming unconventional computers: Dynamics, development, self-reference. *Entropy*, *140*(12), 1939–1952.

Stepney, S., Abramsky, S., Adamatzky, A., Johnson, C. G., & Timmis, J. (2008). Grand challenge 7: Journeys in non-classical computation. In *Visions of Computer Science, London, UK* (pp. 407–421), BCS.

Stepney S, Diaconescu A, Doursat, R., Giavitto, J. -L., Kowaliw, T., Leyser, O., et al. (2012). Gardening cyber-physical systems. In *UCNC 2012*, vol. 7445 of LNCS (pp. 237–238). Berlin: Springer.

Stovold, J., & O'Keefe, S. (2017). Associative memory in reaction-diffusion chemistry (pp. 141–166). In Adamatzky2017v2.

Swade, D. (1995). Charles Babbage's difference engine no. 2: Technical description. Science Museum Papers in the History of Technology 4, September 1995.

Todd, S., & Latham, W. (1992). *Evolutionary art and computers*. New York: Academic Press.

Turing. A.M. (1952). The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society of London. Series B, Biological Sciences*, *2370*(641), 37–72.

Wegner, P. (1997). Why interaction is more powerful than algorithms. *Commun. ACM*, *400*(5), 80–91.

Winfree, E. (2004). DNA computing by self-assembly. In *2003 NAE Symposium on Frontiers of Engineering* (pp. 105–117). Washington, DC: National Academies Press.

Worth, P., & Stepney, S. (2005). Growing music: Musical interpretations of L-systems. In *EvoMusArt 2005*, vol. 3449 of LNCS (pp. 545–550). Berlin: Springer.